ENGR-UH 3511

Computer Organization and Architecture

Name: Nishant Aswani

Net ID: nsa325

Assignment Title: Lab 5

Microprocessor Design and Verilog HDL: Part 3

Nishant Aswani, nsa325@nyu.edu

Computer Organization and Architecture(ENGR-UH 3511), Instructor Cristoforos Vasilatos

1 Introduction

To have the MIPS CPU autonomously determine which datapaths to select, the CPU requires a control unit. The control unit decodes instructions in order to activate certain paths.

The following lab uses Verilog to continue an implementation of a 32-bit MIPS CPU. The modules implemented include a program counter, instruction memory, register file, ALU, data memory, multiple MUXes, half-adders, sign-extension, a control unit, and an ALU control.

2 Methodology

Two new modules, the control logic and ALU control, were added to the top-level module. The control logic was designed to use the opcode section of the instruction to enable or disable certain control wires to carry out expected behavior.

The ALU control provies specific instructions to the ALU given that the ALUOp signal does not differentiate between the various R-type instructions. Test benches were created for both modules to verify expected output.

Next, the `sample.s` file was converted into binary for the instruction memory.

```
mem[0] <= 32'b00000000000000000000000000000000; // EMPTY
mem[1] <= 32'b00100001000010000000000000000010; // addi $t0 $t0 2
mem[2] <= 32'b00100001010010100000000000000010; // addi $t2 $t2 2
mem[3] <= 32'b00000001000010100100000000100000; // add $t0 $t0 $t2
mem[4] <= 32'b00100001001010010000000000000001; // addi $t1 $t1 1
mem[5] <= 32'b10101101001010000000000000000000; // sw $t0 0($t1)
mem[6] <= 32'b10001101001010100000000000000000; // lw $t2 0($t1)
mem[7] <= 32'b00000001000010010100000000100010; // sub $t0 $t0 $t1
mem[8] <= 32'b00010001000010010000000000000001; // beq $t0 $t1 end
mem[9] <= 32'b00001000000000000000000000000111; // j loop
```

The offset for `beq` and the address for `j` were determined by taking into account the order of the instructions. The halt instruction from the assembly code was left out. The program simply jumps to an uninitialized location leading to an erroneous output, implying the end of program.

3 Results

3.1 Control Unit Test Bench

To test the control unit, the opcodes from Table 2 in the lab assignment were cycled through once. Each of the signals corresponds to the expected output shown in the table.

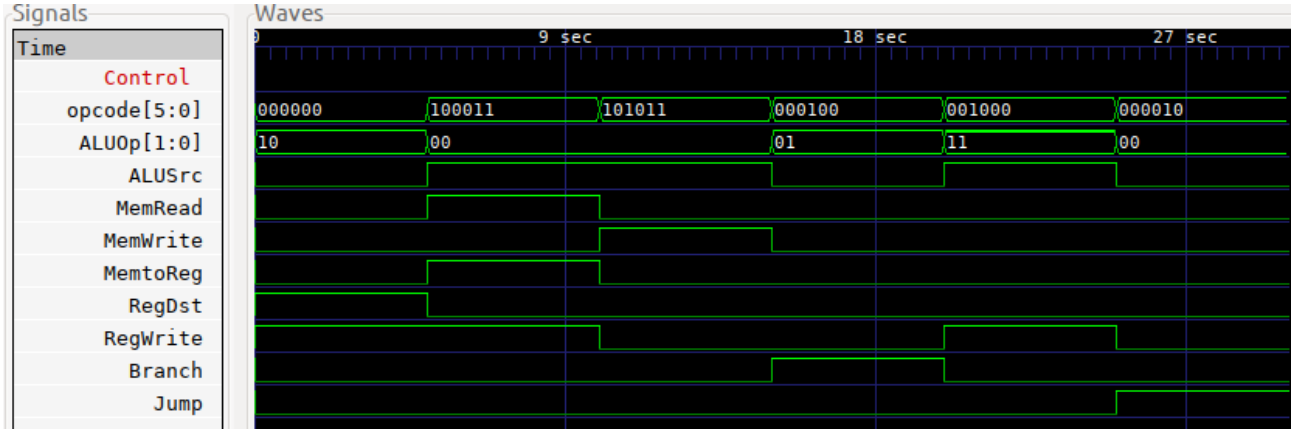

Figure 1: Wave output of control unit test bench

3.2 ALUControl Unit Test Bench

To test the ALUControl, the ALUOp and funct from Table 3 in the lab assignment were cycled through once. Once again, the outputs match that of the given table.

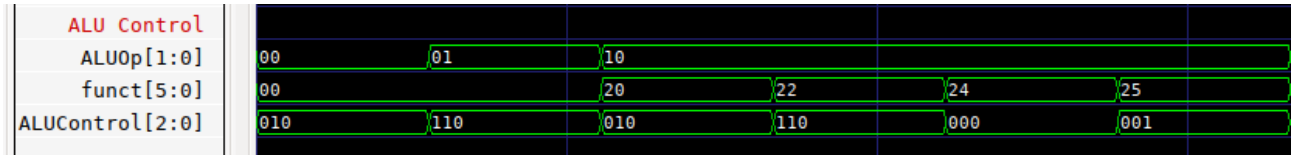

Figure 2: Wave output of control unit test bench

3.3 Sample.s Program

The screenshot below shows the program counter along with the instructions being executed in that cycle. We see that after the PC arrives to address 36, it returns up to address 28. This is representative of the branch condition being unfulfilled and the jump leading the PC back up to address 28.

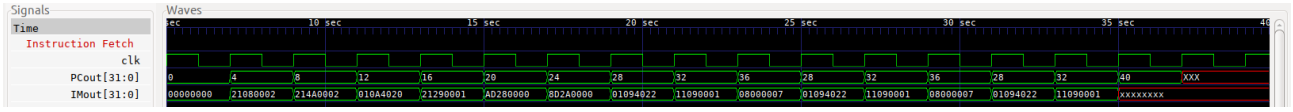

Figure 3: Wave output of program counter and instruction memory output

However, in the second-to-last cycle in the image above, we see that the PC branches from 32 to 40, showing that the branch condition was fulfilled and the program execution was completed.

Next, we take a look at the registers and the values they contain. We see that the registers being accessed correspond with those requested by the instruction. For example, the first instruction requires register `$t0`, or 8, for read and write. The initial value stored is `0`, but we see the `RFWriteData` signal carrying `2` to write into register `$t0`.

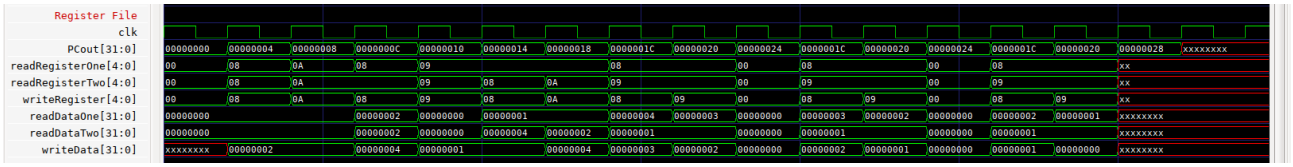

Figure 4: Wave output of registers and register values

Then, we consider the output of the control module, which receives the opcode as an input.

Going in order, we see that the `RegWrite` signal is enabled throughout the program, except for the instructions at addresses `20, 32, 36`, the `sw, beq, and j` instructions, respectively.

We also see that `RegDst` and `ALUSrc` enable on instructions where an alternate source for a signal is required.

The `memRead` signal enables only once for the instruction at address `24`, while `memWrite` and `MemtoReg` signal enable only once for the instruction at address `20`. These correspond the the `lw` and `sw` instructions, respectively.

The `BranchCtrl` signal is enabled at every branch instruction, but we see that the `Branch` signal is enabled only once in the second-to-last cycle.

On the other hand, the jump signal is enabled at every jump instruction, as it is an unconditional jump.

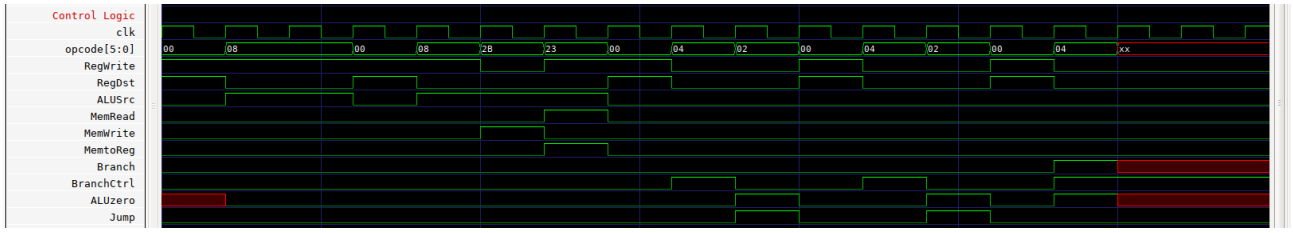

Figure 5: Wave output of the control unit

The ALU is controlled with the `ALUControl` module, which receives input from the funct slice of the instruction and the ALUOp from the control logic.

Here we see that the `ALUControl` unit is able to send the signal for the correct arithmetic operation despite the variety of `funct` and `ALUOp` codes. The first 6 instructions, despite being quite different, all require the `ALU` to carry out addition, which is what `010` implies.

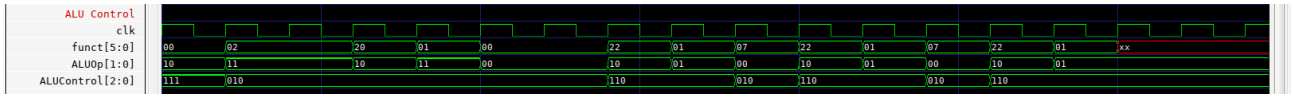

Figure 6: Wave output of the ALU control unit

We also take a look at the ALU to see the outputs of the arithmetic. Looking at the instructions corresponding to `PCOut` at `28` and `32`, we see the `110` `funct` code. Here, the the instruction at `28` is the `sub` instruction carrying out a subtraction to update the value of register `$t0`. The next instruction is the `beq` carrying out a check. We see that the output of the `sub` instruction becomes the input of the next `ALU` operation. We also see that the `zero` flag is raised on the unconditional jump instructions and when `beq` is executed.

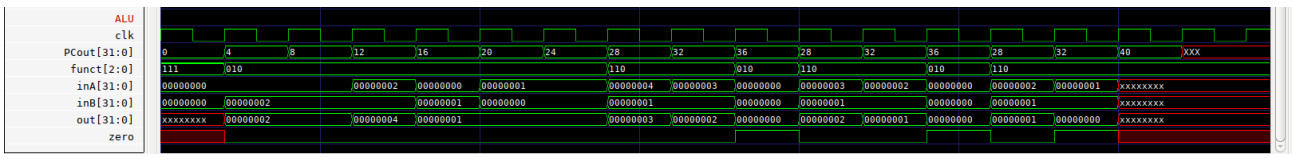

Figure 7: Wave output of the ALU

Finally, we look at the register values. Register`$t1` remains the value `1` it was assigned using the `addi` instruction. Register`$t2` changes from the `lw` instruction, while the data memory updates due to the `sw` instruction. Register`$t0` decrements from `4` to `1`, when it is then equal to `$t1` and the branch executes.

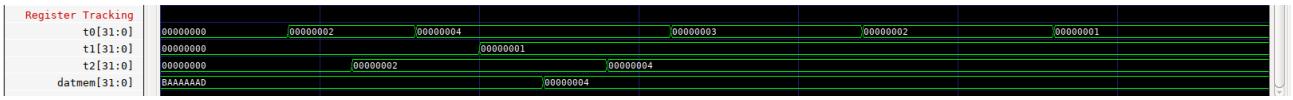

Figure 8: Wave output of registers and data memory

4 Conclusion

This single-cycle implementation of the CPU is now capable of carrying out branch and jump instructions as well as selecting datapaths without hard-coded control logic.